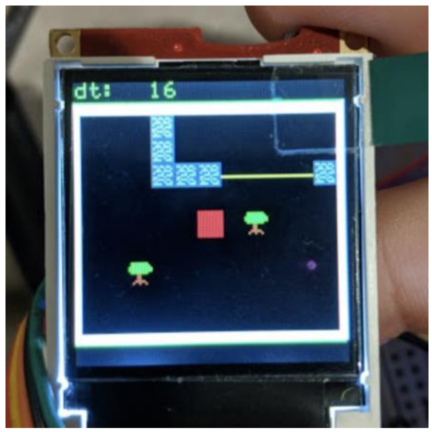
GTLegend: A Role-Playing Game

The goal of this project is to build a Quest game on the Mbed platform, therefore fulfilling your life long goal to be a video game designer. Since the very first video game platforms, top down role-playing games (RPGs) have held an important place in the video game corpus. Such notable titles as *The Legend of Zelda* and *Pokemon Red Version* are exemplars of this style. In this project, you will build a handheld top-down RPG game using the gaming circuit constructed during HW3 and your P2-1 hashtable implementation.



In your top down RPG, the protagonist will be controlled by tilting the game board; the accelerometer will be used as the input for character motion. The buttons will be used to trigger actions in the game. Crucially, the map area of the game will be much larger than the area you are able to display on the screen. Objects on the map will be stored in a hash table, and you will look up the correct locations to display them at every game update.

There are several basic features that your game must implement in order to receive baseline credit; advanced features are an opportunity to earn full and possibly extra credit. The list of basic features and examples of some advanced features are given below. (See rubric for grading details.)

Basic Features

These features are for baseline credit and constitute a functional, but minimal, adventure game. Please note that the story, art, and actions available in the game are intentionally vague, and the specific design of the game is up to your creativity. RPG games are extremely varied, and this project is an opportunity for you to make something unique. Have fun with it!

Player Motion & The Map

The Player character in this top-down RPG moves around in the direction of the tilt of your breadboard, as measured by the accelerometer.

The Map for your game is the world that the character moves around in. The Map is made up of individual tiles, and a grid of 11 x 9 tiles is displayed on the screen at any time. The character is always displayed in the center of the screen. The Map should be at least 50 x 50 tiles, and should have walls around the edges to prevent the Player from leaving the map. The Player should not be able to walk through the walls.

You will need to populate the Map with Items relevant to your game - these Items can include scenery, walls, Non-Player Characters (NPCs), objects, stairs, etc. Implementation details for the Map are discussed in detail in the accompanying technical document.

World Interaction

There are several buttons available in the hardware setup created in HW3. Two of them have required functions:

1. **Action button:** This is the primary way the player interacts with the world. Pressing this button will initiate conversation with NPCs, scroll speech bubbles, and interact with all other objects you might add. The particular action triggered when this is pressed depends on the player's location in the world. For example, pressing the action button while standing near the door (i.e. an object that you might add) would trigger unlocking the door.
2. **Omnipotent-Mode button:** When this button is pressed, "Omnipotent Mode" is toggled. When in Omnipotent Mode, the character should be able to walk anywhere in the world. This is to facilitate grading. For example, if for some reason the door won't unlock even when you're holding the key, your grader can still continue with the quest.

You are free to use the other buttons for any features of your choice. These might be useful to open a menu, for example.

The Quest

The crux of this game, as with all good RPG games, is a quest! In order to win, the Player must complete a quest by interacting with characters and objects in the world in order to find a sonar system that tells the relative location of a treasure chest and win the game.

The quest proceeds in seven steps:

1. Talk to an NPC to start the quest. The NPC instructs the player to use the ladder and go to the second map to find the sonar. Only after that will the ladder be visible to the player.
2. The second map is a maze game and the player should search the map and find the sonar and return to the main map.
3. The player must have the sonar to be able to search for the treasure. After that, by pressing the action button in the main map, a hint will be displayed (in the speech bubble) on the screen that shows the direction of the treasure that is relative to the player's current location. The hint should be in the DirDir format where Dir could be north, south, west, or east (e.g. northeast, southwest, etc..). NOTE: the hint does not tell the exact location, instead, it gives only the direction that the player should go in order to find the treasure. NOTE: The treasure should be included in the main map only after the player grabs the sonar, meaning that the player cannot find the treasure before having the sonar.
4. The player should push the action button on the treasure location to grab the treasure.
5. You must randomly select the location of treasure when the game starts, so every time the game starts, the treasure is in a different location.
6. Treasure moves at least once during game play: Even if the player finds the correct location, the location of the treasure chest should be changed one time and the player must search again. In this case, after pressing action button on the treasure's location, a message must appear in the speech

bubble section, saying that the treasure chest location has been changed to a new hidden location. The sonar should give DirDir hints to the new location.

7. After finding the final treasure chest, display a game over screen.

The purpose of action button is to implement the above functionalities. For example, to begin the quest the player should stand adjacent to the NPC and press the action button, triggering Step 1 and displaying a speech bubble. Speech bubbles should cover the bottom part of the map and should scroll when the player presses the action button. For all other objects that you might add, you can use this button to trigger an interaction with them. More detail on their implementation is given in the technical document. Note that at each point of time, the player should be able enter/exit the second map (the maze map) after talking to the NPC. If the player already has grabbed the sonar, there should be no sonar in the maze map anymore (instead, it is carried with the player).

Graphics

You should put some effort into making your game look good! At least one sprite is required as a basic feature. A sprite in this context is a tile that's more than just a rectangle. You should be using the `uLCD.BLIT` or `draw_img` functions to accomplish this.

A status area is set aside at both the top and bottom of the screen. The top status area should display at a minimum the current player coordinates within the map. These areas can also be used to show information such as quest progress, character inventory, what the action button will do at this location, etc.

For clarity while the Player is moving, it's a good idea to have some background items on the map that scroll by as the player moves. The trees in the demo fulfill this purpose. This is not strictly required, but you'll probably want to add them, since they provide some context while searching for the treasure.

Basic Feature Summary:

1. Accelerometer moves the player.
2. Walls block character motion.
3. Omnipotent-mode button walks through walls, locked doors, etc.
4. The map must be bigger than the screen (at least 50*50 tiles).
5. Quest works (as described above).
6. Display game over screen when quest is complete.
7. Status bar shows player coordinates.
8. Speech bubbles are used in the quest.
9. Art includes at least one sprite.

Advanced Feature Examples

Advanced features in this project are open ended and allow you to make the game your own. The features listed below are all acceptable, but this is not an exhaustive list. Each extra feature is worth +5 points, and you must tell your grader before the demo begins which extra features you used. Other features that are at or above the difficulty level of those listed here are acceptable *but must be approved by an instructor or* *graduate TA*: there will be a pinned discussion topic on Piazza to confirm extra features. If you intend to use features that aren't listed here, start a follow-up on this discussion to clear it with the GTAs or instructors before grading begins.

- **Periodic change of treasure location w/ countdown clock**: The location of the treasure will be changed after a certain period of time. Thus, the direction hints should be updated according to the new location if the player cannot find the treasure chest within that period. When the player has less than T seconds time left (e.g. T=20 seconds). A countdown counter should appear on top of the screen showing the remaining seconds left until the next period begins.
- **Sound cues instead of sonar when get close to treasure**: The sonar will be disabled after the player reaches a certain distance from the treasure chest, instead a beep sound indicates how close the player is to the treasure. The frequency of the beep sound determines how close the player is. NOTE: In this case the treasure should be hidden and there should be an extra button for digging the ground to search for the treasure.
- **Limited sonar use:** The sonar will be broken after pushing action button (excluding the times pressing the action button for interacting with other objects) some number of times. After that, the player should again use the ladder to go to the second map and again find a new sonar in a new location in the maze map. In this case, you should have at least two different maze maps and alternate between these two maps every time the player climbs up the ladder and enters the second map. Also, the location of the sonar within the map should be changed every time the player enters the second map. After the player grabs the sonar, until the next time that the sonar is broken, no sonar is included in the second map, but the second map should be changed (between the two maze maps) every time the player enters the second map. NOTE: it is not necessary to alternate between two maze maps (and update the location of sonar within the map) for the first time the player wants to grab the sonar (i.e. after talking to the NPC).
- **Localized movement of treasure:** The treasure chest will move around a general area as you look for it in addition to jumping across the map when found. In this case, the treasure chest should become hidden once the player is within a certain distance to the treasure.
- **Difficulty modes:** Give the player the option to change the difficulty of the quest. For example, one way of increasing the difficulty is reducing the periods that the treasure chest changes its location if you are planning to implement the first advanced feature. You can think of other ways to set the difficulty as well. But the difficulty should change at least 2 parameters.
- **Sparling treasure:** Scan could be a radius which causes the treasure chest to sparkle on the screen.
- **Fake treasure:** There could be fake treasure chests.
- **Improved tools available:** The treasure chest could be difficult to get with specific tools so you can spend some time to find better tools.
- Add a **start page.**
- **Sound effects** for interactions / background music
- **Different modes of locomotion** (e.g., running, hopping, etc.) They should be visually distinctive.

- **Animation** for interactions with things in the map (e.g., exclamation mark above NPC when you are talking, apple trees look different when apples are picked off etc.).
- **In-game menu**:
  - Save the game
  - Show status information
  - Configuration (Accelerometer direction, which button is which, etc.)
- **Multiple lives** and the possibility to lose:
  - Health & stuff that hurts you.
  - Spikes, enemies, etc.
- **Player can manipulate items** in game: e.g., blowup/destroy a wall.
- **Mobile (walking) NPCs**.
- Player plays against an **intelligent opponent** in a game.
- **Save the game** (persistent over power-off)
- **Multiple quests** (>=2), or extra NPC relevant to the quest(s) (>=5). Only applicable once.
- **Occlusion:** Bigger objects in the map that blocks the character.
  - A very tall tree that hides the character.
  - A feature you can walk behind/under such as a bridge.

P2-2 Technical Reference

In this project, you'll be combining hardware interface libraries for an LCD screen, pushbuttons, speakers, and an SD card reader into a cohesive game. The shell code has several different modules. This document is intended to be a reference for various technical considerations you'll need when implementing your game.

Hash Table

The game will make use of your HashTable library, implemented in P2-1. In order to use this library within the Mbed environment, the easiest strategy is to simply copy and paste the code into the correct files. The shell project has two files already for this purpose: hash_table.cpp and hash_table.h. Copy your completed code from P2-1 into these files before starting anything else.

USB Serial Debug

Debugging is an important part of any software project, and dealing with embedded systems can make debugging difficult. Fortunately, there is a built-in serial monitor on the Mbed that allows you to see printf-style output from the Mbed on your computer. The tutorial to set that up can be found here:

https://os.mbed.com/handbook/SerialPC

The Serial pc object described in the tutorial is already set up for you in globals.h, so you won't need to declare it again. Any file that includes this header can print to the USB like so:

```
pc.printf("Hello, world!\r\n");
```

Game Loop Overview

The basic structure for organizing a video game is called this *game loop*. Each iteration of this loop is known as a *frame*. At each frame, the following operations are performed, typically in this order:

1. Read inputs
2. Update game state based on the inputs
3. Draw the game
4. Frame delay

You'll be implementing parts of each of these steps, along with setting up the game loop to call them in the correct order. The game loop shell code with timing already implemented is in main.cpp.

Read Inputs. Reading user input for a frame happens only once during the frame. This serves two purposes. For one, it isolates the part of the code that has to deal with the input hardware to just the read_inputs function, allowing the rest of the code to deal with only the results of the input operation. Secondly, it ensures a constant value of the "true" input for a particular frame. If there are multiple parts of your game update logic that have to interact with the inputs, it is convenient to know that these inputs are guaranteed to be the same.

Update game. This is where the magic happens. Based on the current state of the game -- where the Player is standing, whether the player is holding the key, what the NPCs are doing -- you compute what the next state should be, based on the inputs you've already measured. For example, if the accelerometer is tilted toward the top of the screen, the Player should move up in the map. This is where most your development will be focused.

Draw game. With the state updated, you now need to show the user what changed by drawing it to the screen. This step is discussed in much more detail below, but for now you'll want to know that the entry point to this portion of the code is called draw_game.

Frame delay. By default, loops in C run as fast as the instructions can possibly execute. This is great when you're trying to sort a list, but it's really bad for games! If the game updates as fast as possible, the user might not be able to understand what's happening or control the character appropriately. So, we introduce a delay that aims to make each frame take 100ms. The time for all the proceeding 3 steps is measured, and the remaining time is wasted before starting again. If more than 100ms has already passed, no additional delay is added. As you're developing your game, be careful that your frames don't get too long, or the feel of your game will degrade.

Map Module

In order to think about updating the game state (moving the player) and drawing the screen, we first need some way to represent the world. This module accomplishes that task.

The map is a two-dimensional grid whose origin is at the top-left corner of the world. The X coordinate increases toward the right, and the Y coordinate increases toward the bottom. This left-handed coordinate system is chosen for consistency with the graphics; see that section for more details. The finest granularity of the map is a single grid cell; the player moves from cell to cell, and each cell contains at most one MapItem. If the cell is empty, then that cell is free space on the map.

The map in this game is represented by a collection of MapItems held in a HashTable. The keys in the HashTable are (x,y) pairs. The data in the HashTable are all of type MapItem, defined in map.h. So, for example, if you access the key "(10, 23)" in the HashTable and the data is a MapItem whose type is WALL, then the player should not be able to walk into that cell.

The shell code is written so that the use of the HashTable is hidden inside the map module; that is, the hash_table.h is only included from map.cpp, and the HashTable functions are only used internally to that module. The public API of the map module does not expose the HashTable, since this is an implementation detail of the map and does not affect the rest of the game. This hides the complexity of the HashTable (questions like "what is the best hash function?" and "how do I map (x,y) pairs into integer keys for use in my hash table?") within the map module itself, and simplifies the rest of the game logic.

The public API for the map module is given in map.h. All functions and structures are documented there. There are functions for accessing items in the map (e.g. get_here, get_north), modifying the map (e.g. add_wall), and selecting the active map. *You are encouraged to add more functionality to this API as you deem necessary for your game.* The point of an API is to be useful to the programmer; if these functions are insufficient, add more!

Map Items. This is the basic unit of the map, and is the underlying type of all the void* data in the map HashTable. Each MapItem has an integer field, type, which tells you what kind of item it is. This allows you to store different information in the map, such as the location of walls and the location of trees, using the same data structure. Each MapItem also has a function pointer of type DrawFunc, that will draw that MapItem. Its inputs are a pixel location (u,v) of the tile. Finally, each MapItem has two additional parameters: an integer flag, walkable, that describes if the player is allowed to walk on that cell; and a void* data for storage of any extra data required during the game update. Walls probably don't need extra data; NPCs or stairs or the door might.

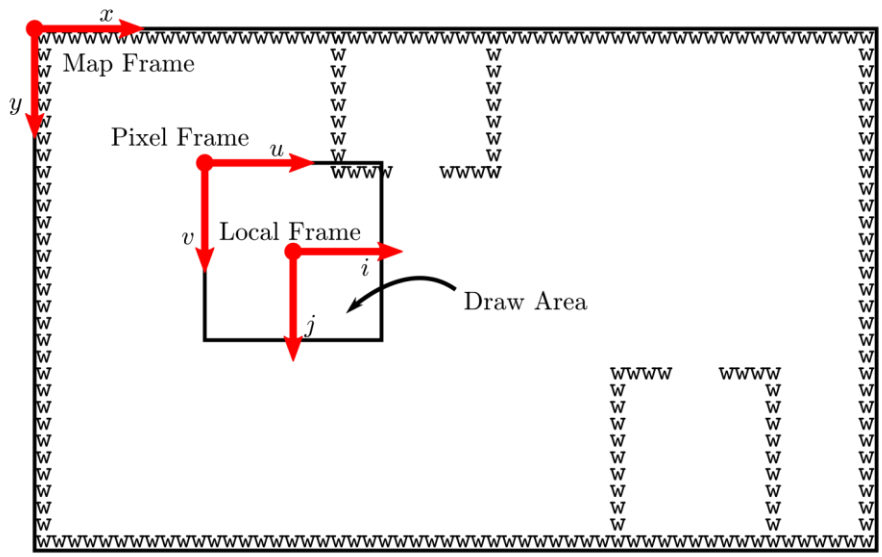
Two-dimensional keys. As you implemented in P2-1, the HashTable accepts only unsigned integers as keys. However, for this application you need to use two integers (the X & Y coordinates) as the key. In order to do this, you need to have a function to map these coordinates unambiguously into a single integer. This function is called XY_KEY, and is private to map.cpp. You then also need a hash function that will take this key as normal and produce a hash value for bucket selection in the HashTable.

The Active Map. All operations in the Map API use the "active map." In the shell code, there is only one active map. The function get_active_map returns this map, and the function set_active_map does nothing. You may modify these to allow selection between multiple maps. Once an active map is selected using set_active_map, all other functions (accessors and modifiers) will use the currently active map. Only one map can be active at a time; setting a new active map implicitly deactivates the previous active map.

Graphics

The graphics module houses most of the drawing code for the game. This includes all the drawing functions for the various MapItems. The entry point for drawing the screen under normal operations (not in a speech bubble) is the draw_game function. This section describes how that function accomplishes drawing the tiles, and various ideas to consider as you extend this function for your own game.

Coordinate Reference Frames. There are several relevant coordinate frames for this game. The first we have already covered: the map frame. This frame's coordinates are labelled (x,y) and its origin is the top corner of the map. X increases right, and Y increases down. All frames in the game are this left-handed orientation.



The next frame is the local drawing frame. This frame is centered on the Player, and ranges from (-5,-4) to (5,4), i.e. it is an 11x9 grid of cells. This frame is iterated in draw_game and each cell in drawn in turn using the DrawFunc from the map, or a draw_nothing function if there is no MapItem. The coordinates of this frame are labelled (i, j).

Finally, there are the pixels on the screen. This frame has its origin at the top-left corner of the screen. The screen is at 128x128 array of pixels. The coordinates of this frame are labelled (u,v). Each cell in the map is 11x11 pixels.

Drawing functions. As noted above, each MapItem has an associated DrawFunc that knows how to draw that item. These functions take as input a (u,v) coordinate for the top-left pixel of the tile, and draw an 11x11 image that represents the MapItem. An example is the draw_wall function, in graphics.h.

You will need to add more draw functions as you add more types of MapItem. You are free to implement these in whatever way you like, using the full power of the uLCD library. However, a simple way to do this has been given to you. The draw_img function takes a string of 121 (= 11 * 11) characters, each representing a pixel color, and translates that into a BLIT command to draw those colors to the screen. You can use this function to make nice graphics very simply by defining a new string that represents the image you want to draw. This is the recommended method for generating art for your game.

Drawing Performance. The screens are notoriously slow to draw, and the length of time it takes to complete a drawing command is proportional to the number of pixels that it changes on the screen. So, the drawing code goes through some hoops to make sure that things keep moving quickly. In particular, the drawing code requires not only the current player position (Player.x and Player.y) but also the previous position (Player.px and Player.py), in order to determine what has changed on the screen. If an element on the screen has not changed, it is not redrawn. This saves time and make the game update more quickly. You'll need to be careful with this as you decide how many items to put in your map and how to draw the new items you add.

You must design, implement, and test your own code. There are many, many ways to code this project, and many different possibilities for timing, difficulty, responsiveness and general feel of the game. Your project should represent your interpretation of how the game should feel and play. Any submitted project containing code (other than the provided framework code and mbed libraries) not fully created and debugged by the student constitutes academic misconduct.

Mbed reference materials: http://ece2035.ece.gatech.edu/readings/embedded/index.html